Mining Unstructured Data

11. Transformers

Javier Ferrando

Based on Stanford CS224N and The Illustrated Transformer
Recap
Language Modeling

- Language Modeling is the task of predicting what word comes next:

The students opened their ________

books

laptops

exams

\[ P(x^{(t+1)} \mid x^{(t)}, \ldots, x^{(1)}) \]
N-gram Language Models

Suppose we are learning a 4-gram Language Model.

\[ P(w|\text{students opened their}) = \frac{\text{count(students opened their } w)}{\text{count(students opened their)}} \]
A RNN Language Model

output distribution
\[ \hat{y}(t) = \text{softmax} \left( U h^{(t)} + b_2 \right) \in \mathbb{R}^{|V|} \]

hidden states
\[ h^{(t)} = \sigma \left( W_h h^{(t-1)} + W_e e^{(t)} + b_1 \right) \]

\( h^{(0)} \) is the initial hidden state

word embeddings
\[ e^{(t)} = E x^{(t)} \]

words / one-hot vectors
\[ x^{(t)} \in \mathbb{R}^{[V]} \]
Long Short-Term Memory RNNs (LSTMs)

**Forget some cell content**

Compute the forget gate

**Compute the input gate**

Compute the new cell content

Compute the output gate

Write some new cell content

Output some cell content to the hidden state
Index

- Sequence-to-sequence Models
  - Neural Machine Translation
  - Decoding Strategies

- Attention

- Transformer
  - Issues with RNNs
  - Self-attention
  - Multi-head attention
  - Positional Encodings
  - Decoder
Sequence-to-sequence Models
Sequence-to-sequence tasks

Many NLP tasks can be phrased as sequence-to-sequence:

- **Summarization** (long text -> short text)
- **Dialogue** (previous utterances -> next utterance)
- **Parsing** (input text -> output parse as sequence)
- **Code generation** (Natural Language -> Python Code)
- **Translation** (source sentence -> translation)
Sequence-to-sequence model
Sequence-to-sequence model

Encoder RNN produces an encoding of the source sentence.

Decoder RNN is a Language Model that generates target sentence, conditioned on encoding.

Note: This diagram shows test time behavior: decoder output is fed in \( \cdots \cdots \rightarrow \) as next step's input.
Neural Machine Translation

- The sequence-to-sequence model is an example of a Conditioned Language Model
  - Language Model because the decoder is predicting the next word of the target sentence $y$
  - Conditioned because its predictions are also conditioned on the source sentence $x$

- NMT computes $P(y|x)$:
  \[
P(y|x) = P(y_1|x) P(y_2|y_1, x) P(y_3|y_1, y_2, x) \ldots P(y_T|y_1, \ldots, y_{T-1}, x)
  \]

- We train these models with a parallel corpus
Training a Neural Machine Translation system

\[ J = \frac{1}{T} \sum_{t=1}^{T} I_t \]

- \( J_1 \) = negative log prob of "he"
- \( J_2 \) = negative log prob of "with"
- \( J_3 \) = negative log prob of \(<END>\)

Source sentence (from corpus):
- \( il \)
- \( a \)
- \( m' \)
- \( entarté \)

Target sentence (from corpus):
- \(<START>\)
- \( he \)
- \( hit \)
- \( me \)
- \( with \)
- \( a \)
- \( pie \)

Encoder RNN

Decoder RNN

Teacher forcing
Greedy decoding

At each timestep we take the most probable word (argmax).

Decoder RNN is a Language Model that generates target sentence, conditioned on encoding.

Note: This diagram shows test time behavior; decoder output is fed in \( \cdots \) as next step's input.
Exhaustive search decoding

- We want to find the translation $y$ that maximizes:

$$P(y|x) = P(y_1|x) P(y_2|y_1, x) P(y_3|y_1, y_2, x) \ldots, P(y_T|y_1, \ldots, y_{T-1}, x)$$

$$= \prod_{t=1}^{T} P(y_t|y_1, \ldots, y_{t-1}, x)$$

- We could achieve the optimal translation by tracking all possible sequences
  - This means that on each step $t$, we track $V^t$ possible partial translations, where $V$ is the vocabulary size
  - This $O(V^T)$ complexity is too expensive
Beam search decoding

- On each step of decoder, keep track of the k most probable partial translations (which we call hypotheses)
  - K is the beam size (in NMT, around 5 to 10)

- Each hypothesis has a score, its log probability:
  \[
  \text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)
  \]
  - We search for high-scoring hypotheses, tracking top k on each step

- Beam search doesn’t guarantee an optimal solution, but is more efficient than exhaustive search
Beam search decoding: example

Beam size $k = 2$

$$\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)$$

<START>
Beam search decoding: example

Beam size $k = 2$

$$\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)$$

-0.7 = log $P_{LM}(he|\text{<START>})$

-0.9 = log $P_{LM}(l|\text{<START>})$

Take top $k$ words and compute scores
Beam search decoding: example

Beam size $k = 2$

For each of the $k$ hypotheses, find top $k$ next words and calculate scores
Beam search decoding: example

Beam size $k = 2$

$$\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i|y_1, \ldots, y_{i-1}, x)$$

Of these $k^2$ hypotheses, just keep $k$ with highest scores.
Beam search decoding: example

Beam size $k = 2$

$$\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)$$

For each of the $k$ hypotheses, find top $k$ next words and calculate scores.
Beam search decoding: example

Beam size $k = 2$

$$\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)$$

Of these $k^2$ hypotheses, just keep $k$ with highest scores
Beam search decoding: example

Beam size $k = 2$

\[
score(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)
\]

For each of the $k$ hypotheses, find top $k$ next words and calculate scores.
Beam search decoding: example

Beam size $k = 2$

$$\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)$$

Of these $k^2$ hypotheses, just keep $k$ with highest scores
Beam search decoding: example

Beam size $k = 2$

$$\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)$$

Diagram showing the beam search decoding process with examples of words and their scores.
Beam search decoding: stopping criterion

- **In greedy decoding**, usually we decode until the model produces an `<END>` token
  - `<START>` he hit me with a pie `<END>`

- **In beam search decoding**, different hypotheses may produce tokens on different timesteps
  - When a hypothesis produces `<END>`, that hypothesis is complete
  - Place it aside and continue exploring other hypotheses via beam search

- Usually we continue beam search until:
  - We reach timestep $T$ (where $T$ is some pre-defined cutoff), or
  - We have at least $n$ completed hypotheses (where $n$ is pre-defined cutoff)
Beam search decoding: selecting the best hypothesis

- Each hypothesis in our list of hypotheses has a score

\[
\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)
\]

- Problem with this: longer hypotheses have lower scores

- Fix: Normalize by length. Use this to select top one instead:

\[
\frac{1}{t} \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)
\]
About the success of NMT

Neural Machine Translation went from a fringe research attempt in 2014 to the leading standard method in 2016

- 2014: First seq2seq paper published
- 2016: Google Translate switches from SMT to NMT – and by 2018 everyone has
NMT is far from solved

- NMT picks up biases in training data
NMT is far from solved

- Hard to interpret systems do strange things
Sequence-to-sequence: the bottleneck problem
Attention
Attention

- Attention provides a solution to the bottleneck problem.

- Core idea: on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence
Attention

On this decoder timestep, we’re mostly focusing on il (he)

Dot product

Sometimes called the context vector

Use the attention distribution to take a weighted sum of the encoder hidden states.

The attention output mostly contains information from the hidden states that received high attention.

Source sentence (input)
Attention

![Diagram showing the process of attention in Transformers]

- **Encoder RNN**: Input sequence (il, o, m', entarté) is processed by the RNN to generate encoder hidden states.
- **Attention**: The attention mechanism computes attention weights which are then used to weight the encoder hidden states.
- **Attention distribution**: The weighted sum of encoder hidden states is computed to produce the attention output.
- **Decoder RNN**: The output of the attention mechanism is concatenated with the decoder hidden state (h_e) to compute the next decoder hidden state (h_{t+1}).

The process involves:
- Concatenating attention output with decoder hidden state, then using to compute $h_{t+1}$ as before.
Attention
Attention
Attention
Attention
Attention

Source sentence (input)

Transformers
Attention equations

• We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
• On timestep $t$, we have decoder hidden state $s_t \in \mathbb{R}^h$
• We get the attention scores $e^t$ for this step:
  $$e^t = [s_t^T h_1, \ldots, s_t^T h_N] \in \mathbb{R}^N$$
• We take softmax to get the attention distribution $\alpha^t$ for this step (this is a probability distribution and sums to 1)
  $$\alpha^t = \text{softmax}(e^t) \in \mathbb{R}^N$$
• We use $\alpha^t$ to take a weighted sum of the encoder hidden states to get the attention output $a_t$
  $$a_t = \sum_{i=1}^N \alpha^t_i h_i \in \mathbb{R}^h$$
• Finally we concatenate the attention output $a_t$ with the decoder hidden state $s_t$ and proceed as in the non-attention seq2seq model
  $$[a_t; s_t] \in \mathbb{R}^{2h}$$

Several variants for computing attention scores, this one is the dot product attention
Generalization of attention

- We can use attention in many architectures (not just seq2seq) and many tasks (not just MT)

- General definition of attention:
  - Given a set of vector values and keys, and a vector query, attention is a technique to compute a weighted sum of the values, dependent on the query and keys.

- Intuition:
  - The weighted sum is a selective summary of the information contained in the values, where the query and keys determine which values to focus on.
  - Attention is a way to obtain a fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).
Attention

Keys and values

Source sentence (input)

query

Attention output
Advantages of attention

- Attention significantly improves NMT performance
- Attention provides more “human-like” model of the MT process
  - We look back at the source sentence while translating, rather than remembering it all
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with the vanishing gradient problem
  - Provides shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we see what the decoder was focusing on
  - The network learns alignment by itself
Transformer
Attention is all you need

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@gmail.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez†
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaiser@gmail.com

Illia Polosukhin‡
illia.polosukhin@gmail.com

https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
Transformers are used everywhere

- Image Classification [Dosovitsky et al. 2021]
- Protein Folding [Jumper et al. 2021]
- Autonomous cars (Tesla)
Issues with RNNs: Linear interaction distance

- RNNs encode linear locality: useful since nearby words often affect each other’s meaning

- Problem: RNNs take $O(\text{sequence length})$ steps for distant word pairs to interact
  - Hard to learn long-distance dependencies (vanishing gradient problem)
  - Meaning in sentences doesn’t necessarily follow a ‘linear order’
Issues with RNNs: Lack of parallelizability

- Forward and backward passes have $O(\text{sequence length})$ unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can’t be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!
Self-attention

- To recap, **attention** treats each word’s representation as a **query** to access and incorporate information from a **set of values**.
  - **Self-attention** is **encoder-encoder** (or decoder-decoder) attention where each word attends to each other word **within the input (or output)**.

**Transformer Advantages:**
- Number of unparallelizable operations does not increase with sequence length.
- Each "word" interacts with each other, so maximum interaction distance: $O(1)$. 
Transformer

- Self-attention
- Encoder-decoder attention / Cross-attention
- Masked self-attention

- Transformer components:
  - Input Embedding
  - Positional Encoding
  - Add & Norm
  - Feed Forward
  - Multi-Head Attention
  - Outputs (shifted right)
  - Softmax
  - Linear
  - Add & Norm
  - Feed Forward
  - Multi-Head Attention
  - Masked Multi-Head Attention
  - Outputs
Transformer
Encoder: self-attention
Encoder: self-attention

- Recall: Attention operates on queries, keys, and values
  - We have some queries \( q_1, q_2, \ldots, q_T \). Each query is \( q_i \in \mathbb{R}^d \)
  - We have some keys \( k_1, k_2, \ldots, k_T \). Each key is \( k_i \in \mathbb{R}^d \)
  - We have some values \( v_1, v_2, \ldots, v_T \). Each value is \( v_i \in \mathbb{R}^d \)

- In self-attention, the queries, keys, and values are drawn from the same source.

- The (dot product) self-attention operation is as follows:
  \[
  e_{ij} = q_i^T k_j \\
  \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})} \\
  \text{output}_i = \sum_j \alpha_{ij} v_j
  \]
  - Compute key-query affinities
  - Compute attention weights from affinities (softmax)
  - Compute outputs as weighted sum of values
Recipe for Self-Attention in the Transformer Encoder

- Step 1: For each word, calculate its query, key, and value
  \[ q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i \]

- Step 2: Calculate attention score between query and keys
  \[ e_{ij} = q_i^T k_j \]

- Step 3: Take the softmax to normalize attention scores
  \[ \alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})} \]

- Step 4: Take a weighted sum of values
  \[ \text{Output}_i = \sum_j \alpha_{ij} v_j \]
Multi-head self-attention
Multi-head self-attention

• What if we want to look in multiple places in the sentence at once?
  • For word $i$, self-attention “looks” where $x_i^T Q^T K x_j$ is high, but maybe we want to focus on different $j$ for different reasons?
• We’ll define multiple attention “heads” through multiple $Q, K, V$ matrices
• Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where $h$ is the number of attention heads, and $\ell$ ranges from 1 to $h$.
• Each attention head performs attention independently:
  • $output_\ell = \text{softmax}(X Q_\ell K_\ell^T X^T) \ast XV_\ell$, where $output_\ell \in \mathbb{R}^{d/h}$
• Then the outputs of all the heads are combined!
  • $output = Y[output_1; \ldots; output_h]$, where $Y \in \mathbb{R}^{d \times d}$
Encoder: Feedforward

- **Problem**: Since there are no element-wise non-linearities, self-attention is simply performing a re-averaging of the value vectors.
- **Easy fix**: Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

\[
m_t = MLP(output_t) = W_2 \ast \text{ReLU}(W_1 \times output_t + b_1) + b_2
\]
Training tricks

- Training Trick #1: Residual Connections
- Training Trick #2: LayerNorm
- Training Trick #3: Scaled Dot Product Attention
Training trick 1: Residual Connections

- **Residual connections** are a trick to help models train better.
  - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where $i$ represents the layer)
  
  \[
  X^{(i-1)} \xrightarrow{\text{Layer}} X^{(i)}
  \]
  
  - We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$

  \[
  X^{(i-1)} \xrightarrow{\text{Layer}} + X^{(i)}
  \]

- Residual connections are thought to make the loss landscape considerably smoother (thus easier training!)
Training trick 2: Layer Normalization

- **Layer normalization** is a trick to help models train faster.

- Cut down on uninformative variation in hidden vector values by normalizing to zero mean and unit standard deviation within each layer.

\[
\text{output} = \frac{x - \mu}{\sqrt{\sigma + \epsilon}} \cdot \gamma + \beta
\]

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias
Training trick 3: Scaled Dot Product Attention

- The dot product in the attention tends to take on extreme values, as its variance scales with dimensionality $d_k$.

Updated Self-Attention Equation:

$$\text{Output} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$
Fixing the sequence order problem

- Since self-attention doesn’t build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each sequence index as a vector $p_i \in \mathbb{R}^d$, for $i \in \{1, 2, \ldots, T\}$ are position vectors.
- Easy to incorporate this info into our self-attention block: just add the $p_i$ to our inputs!
- Let $\tilde{v}_i, \tilde{k}_i, \tilde{q}_i$ be our old values, keys, and queries.

$$
\begin{align*}
  v_i &= \tilde{v}_i + p_i \\
  q_i &= \tilde{q}_i + p_i \\
  k_i &= \tilde{k}_i + p_i
\end{align*}
$$

We could concatenate instead
Position representation vectors through sinusoids

\[ p_i = \begin{cases} 
\sin(i/10000^{2+1/d}) \\
\cos(i/10000^{2+1/d}) \\
\vdots \\
\sin(i/10000^{2+d/d}) \\
\cos(i/10000^{2+d/d}) 
\end{cases} \]

- It allows to extrapolate (in theory) to longer sequences as periods restart. However, doesn’t work in practice
- Not learnable parameters
Position representation vectors learned from scratch

- **Learned position representations**: Let all $p_i$ be learnable parameters!
  - Learn a matrix $P \in \mathbb{R}^{d \times T}$

- **Pros**:
  - Flexibility: each position gets to be learned to fit the data

- **Cons**:
  - Can’t extrapolate to indices outside 1, ..., $T$

- Most systems use this
Solution: Inject Order Information through Positional Encodings!
Fixing the decoder problem: Masked attention

● **Problem**: How do we keep the decoder from "cheating"? If we have a language modeling objective, can't the network just look ahead and "see" the answer?

● **Solution**: Masked Multi-Head Attention. At a high-level, we hide (mask) information about future tokens from the model.
Fixing the decoder problem: Masked attention

- To use self-attention in decoders, we need to ensure we can’t peek at the future.

- At every timestep, we could change the set of keys and queries to include only past words (Inefficient!)

- To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$

$$e_{ij} = \begin{cases} q_i^T k_j, j < i \\ -\infty, j \geq i \end{cases}$$
Decoder: cross-attention

- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, the cross-attention is in charge of looking for information from the source sequence (like the attention in RNNs).
- Let \( h_1, \ldots, h_T \) be output vectors from the Transformer encoder, \( x_i \in \mathbb{R}^d \).
- Let \( z_1, \ldots, z_T \) be input vectors from the Transformer decoder, \( z_i \in \mathbb{R}^d \).
- Then keys and values are drawn from the encoder:
  \[
  k_i = Kh_i, \quad v_i = Vh_i
  \]
- And the queries are drawn from the decoder, \( q_i = Qz_i \).
Decoder

- Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)
- Add a final softmax to generate a probability distribution of possible next words
Recap of Transformer Architecture